

510.84

I 46r

no. 1034-1041

1980

copy 3

inc.



670.84
T. L. G.
no. 1046

Report No. UIUCDCS-R-80-1040

UILU-ENG 80 1740

AUTOMATIC CELL GENERATION FOR RECURRENCE STRUCTURES

by

Avinoam Bilgory and Daniel D. Gajski

November 1980

NSF-OCA-MCS76-81686-000052



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS



Digitized by the Internet Archive
in 2013

<http://archive.org/details/automaticcellgen1040bilg>

Report No. UIUCDCS-R-80-1040

Automatic Cell Generation for Recurrence Structures^{*}

by

Avinoam Bilgory and Daniel D. Gajski

November 1980

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

^{*}This work was supported in part by the National Science Foundation under Grant No. US NSF MCS76-81686.

Abstract

The cell generation module of a Gate-to-Silicon compiler is described. In this paper the cell generation is restricted to Boolean recurrence structures which appear to be the most difficult for synthesis. The fastest solution of the recurrence can be achieved by the semigroups approach in time proportional to the logarithm of the recurrence length. The solution is accomplished by a network that requires up to 4 different types of cells. Given a Boolean recurrence of any order, the cell generator module generates the Boolean equations of these cells.

Index Terms:

Gate compilers

Logic-design automation

Boolean-recurrence solvers

Table of Contents

| <u>Section</u> | <u>Page</u> |
|--|-------------|
| 1. Introduction | 1 |
| 2. Definitions and Basic Ideas | 5 |
| 3. Network Structure for Serial Solution | 13 |
| 4. Network Structure for Parallel Solution | 21 |
| 5. The Programs and their Role | 33 |
| 5.1. PREPRC | 34 |
| 5.2. BINREC | 42 |
| 5.3. MINIMZ | 50 |
| 6. Examples | 55 |
| List of References | 68 |

1. Introduction

This document describes the cell generation module of a gate-to-silicon compiler.

Such a compiler takes as an input a functional (behavioral) description of a logic module or unit and environmental and technological constraints. The former are exemplified by time delay from inputs to outputs of the module or area of silicon to be occupied by the module in the final layout of the chip, and the latter specify the rules of the process used to manufacture the chip like minimum line width, line capacitances etc. The input description is in a high-level language in which only Boolean variables, scalars and arrays are allowed. For example, a 32-bit binary adder can be described as follows:

```
S1:    C(0) = CIN
        DO I = 1,32

S2:    C(I) = A(I)*B(I) + (A(I) + B(I))*C(I-1)
        END
        DO I = 1,32

S3:    S(I) = A(I) ⊕ B(I) ⊕ C(I-1)
        END
```

The above description can be used for variety of implementation styles. For example, if the delay time specified is relatively slow with respect to technology used the 32-bit adder will be implemented as a ripple-carry adder. If a faster version is required the look-ahead-carry adder will be used. For different delay times different number of bits will be looked ahead. Similarly, different layouts will be produced for different time delays.

The compiler consists basically of four modules:

1. Boolean Analyzer partitions the input description into blocks with easily recognizable structure. For example, the statements S_1 and S_2 will be recognized as a recurrence system while the statement S_3 is detected to be a vector operation. The Boolean Analyzer performs certain transformations on the initial description to achieve an optimal partitioning of statements. For example, if statements S_2 and S_3 are inside one DO loop the Boolean Analyzer will distribute the DO loop into two as given above.

2. Cell Generator consists of several submodules, each for one type of a block recognized by the Boolean Analyzer. Each submodule generates the functional description of the basic cells used to synthesize the given block. For the statement S_3 the basic cell is described by the equation $S = A \oplus B \oplus C$. On the other hand there will be three basic

cells generated for the recurrence block. Namely,

CELL 1: $P = A*B, \quad Q = A + B$

CELL 2: $P = P1 + Q1*P2, \quad Q = Q1*Q2$

CELL 3: $X = P + Q*X0$

3. Silicon Compiler generates automatically the stick diagram of each cell and after compaction translates it into a layout using rules of technology chosen. Each cell can be manually designed, if desired. Such a manual design can be selectively applied to complex or large cells. The electrical and geometrical parameters of each cell like width, height and sizes of devices are passed to the next module.

4. Structure Generator attempts to obtain the best possible structure for the given functional description and environmental parameters. It specifies the number of different cell types, the position of each cell in the final layout and the interconnections between the cells.

In this paper the cell generation is restricted to recurrence structures which appear to be the most difficult for synthesis. A Boolean recurrence of any order can be solved serially in time proportional to the recurrence length n using n basic cells. The parallel solution on the other hand generates the result in time proportional to

$\log n$ with area proportional to $n \log n$. In this case the recurrence solver may require up to four different types of cells which are called nodes in the text that follows.

Section 2 describes the basic idea and introduces some definitions. Section 3 describes the well known serial solution to Boolean recurrences, while section 4 introduces the parallel solution. Section 5 is a description of the programs that generate the nodes' equations, then examples are brought in section 6.

2. Definitions and Basic Ideas

The basic ideas presented in this section appear in [1], and more detailed in [2], section 2.

A recurrence system of the first order $R(1)$ is a quadruple $\langle K, X, x_0, F \rangle$, where $K = K_1 \times K_2 \times \dots \times K_s$ is the Cartesian product of sets of coefficients, $X = X_1 \times X_2 \times \dots \times X_t$ is the Cartesian product of sets of variables, $x_0 \in X$ is the initial value and $F = \{f_k: X \rightarrow X \mid k \in K\}$. Usually, the set F is given in a compound form as the recurrence expression. For example, if A, B, C, D and X are sets of real numbers, then $R_1(1) = \langle A \times B \times C \times D, X, x_0, F_1 \rangle$ may have $F_1: x_i = (a_i + b_i x_{i-1}) / (c_i + d_i x_{i-1})$, with $+$, juxtaposition and $/$ denoting addition, multiplication and division of real numbers.

For Boolean recurrences which are the subject of this report, we will always have $K_p = \{0, 1\}$ for all p , $1 \leq p \leq s$, and $X_q = \{0, 1\}$ for all q , $1 \leq q \leq t$; also $F: x_i = f_i(x_{i-1})$ in a form of Boolean equations.

A Boolean recurrence system of the first order is denoted $B(1)$. For example, $B_2(1) = \langle A \times B, X, x_0, F_2 \rangle$ where $F_2: x_i = a_i + b_i \bar{x}_{i-1}$. In this case, $+$, juxtaposition and $\bar{}$ denote Boolean operators OR, AND and NOT.

The functions in the set F can be extended to sequences of coefficients, so that for all $k_i, k_{i-1}, \dots, k_1 \in K$,

$$f_{k_i, k_{i-1}, \dots, k_1}(x) = f_{k_i}(f_{k_{i-1}, \dots, k_1}(x)).$$

Then the solution of the recurrence system $R(1)$ of length n , denoted by $R\langle n, 1 \rangle$, is the sequence

$$x_n = f_{k_n, \dots, k_1}(x_0)$$

$$x_{n-1} = f_{k_{n-1}, \dots, k_1}(x_0)$$

.

.

.

$$x_1 = f_{k_1}(x_0)$$

for given $k_n, k_{n-1}, \dots, k_1 \in K$ and $x_0 \in X$. Using the associativity rule, we get for all i , $1 \leq i \leq n$:

$$x_i = f_{k_i, k_{i-1}, \dots, k_1}(x_0)$$

$$= f_{k_i}(f_{k_{i-1}}(\dots f_{k_1}(x_0) \dots))$$

$$= (f_{k_i} \circ f_{k_{i-1}} \circ \dots \circ f_{k_1})(x_0)$$

$$= f_{k'_i}(x_0)$$

where the symbol \circ denotes the composition of functions.

Let F^+ denote the set of all functions generated from F under functional compositions. Then $\langle F^+, \circ \rangle$ is the recurrence semigroup.

Every function (called mapping later in the text) f_{k_i} is given a unique binary code, which is called the mapping code. The code assignment has a great influence on the evaluation complexity of $f_{k'_i}$.

The solution of every recurrence system therefore can be decomposed into three subproblems:

- (a) The computation of the functional composition $f_{k'_i} = f_{k_i} \circ f_{k_{i-1}} \circ \dots \circ f_{k_2} \circ f_{k_1}$ for all i , $1 \leq i \leq n$.
- (b) The assignment of a binary code for each f_{k_i} in such a way that the computation of $f_{k_i} \circ f_{k_j}$ is as simple as possible in terms of hardware realization.

(c) Functional evaluation of $x_i = f_{k_i'}(x_0)$ for all i , $1 \leq i \leq n$.

Each of the above subproblems can be solved using only one type of node:

Subproblem (a) can be solved by a tree-like network, consisted of identical nodes, called tree-nodes. Each tree-node takes two functions f_{k_i} and f_{k_j} as inputs and generates their composition $f_{k_i} \circ f_{k_j}$. The shortest solution time is proportional to $\log n$. The function of the tree-node is defined by the recurrence expression, while its complexity depends also on the code assigned to each f_{k_i} .

Subproblem (b) is solved using some heuristic methods, as will be explained in detail in section 5.3. The code assignment defines the pre-node, which transforms the coefficient k_i into the mapping code of f_{k_i} . The transformation is done by a vector consisted of n pre-nodes, in constant time.

Subproblem (c) is also solved by a vector operation in constant time. The vector consists of n identical nodes, called out-nodes. Each out-node takes a function $f_{k_i'}$ and its argument x_0 as inputs, and generates $f_{k_i'}(x_0)$.

The same procedure applies to recurrence of order $m > 1$, after it is transformed into a first order recurrence.

A recurrence system of m -th order $R(m)$ is a quadruple $\langle K, X, \underline{x}_0, F \rangle$ where $\underline{x}_0 = (x_0, x_{-1}, \dots, x_{-m+1})$ is a vector of initial values and $F = \{f_k: X^m \rightarrow X \mid k \in K\}$. The transformation of $R(m)$ into $R(1)$ is done by grouping each m x_i -s together, as well as each m f_{k_i} -s, in the following manner:

$$\underline{x}_m = (x_m, x_{m-1}, \dots, x_2, x_1)$$

$$\underline{x}_{2m} = (x_{2m}, x_{2m-1}, \dots, x_{m+2}, x_{m+1})$$

.

.

.

$$\underline{x}_{im} = (x_{im}, x_{im-1}, \dots, x_{(i-1)m+2}, x_{(i-1)m+1})$$

.

.

.

and

$$f_{k_m, k_{m-1}, \dots, k_2, k_1} = (f_{k'_m}, f_{k'_{m-1}}, \dots, f_{k'_2}, f_{k'_1})$$

$$f_{k_{2m}, k_{2m-1}, \dots, k_{m+2}, k_{m+1}} = (f_{k'_{2m}}, f_{k'_{2m-1}}, \dots, f_{k'_{m+2}}, f_{k'_{m+1}})$$

.

.

.

$$f_{k_{im}, k_{im-1}, \dots, k_{(i-1)m+2}, k_{(i-1)m+1}} = (f_{k'_{im}}, f_{k'_{im-1}}, \dots, f_{k'_{(i-1)m+2}}, f_{k'_{(i-1)m+1}})$$

.

.

.

where

$$f_{k'_{(i-1)m+1}} = f_{k_{(i-1)m+1}}$$

$$f_{k'_{(i-1)m+2}} = f_{k_{(i-1)m+2}} \circ f_{k_{(i-1)m+1}}$$

.
.
.

$$f_{k'(i-1)m+j} = f_{k(i-1)m+j} \circ f_{k(i-1)m+j-1} \circ \dots \circ f_{k(i-1)m+2} \circ f_{k(i-1)m+1}$$

.
.
.

$$f_{k'_{im-1}} = f_{k_{im-1}} \circ f_{k_{im-2}} \circ \dots \circ f_{k(i-1)m+2} \circ f_{k(i-1)m+1}$$

$$f_{k'_{im}} = f_{k_{im}} \circ f_{k_{im-1}} \circ \dots \circ f_{k(i-1)m+2} \circ f_{k(i-1)m+1}.$$

Each individual component of \underline{x}_{im} is given by

$$x_{(i-1)m+j} = f_{k'(i-1)m+j}(\underline{x}_{(i-1)m})$$

for all j , $1 \leq j \leq m$, thus the transformed recurrence is expressed by

$$\underline{x}_{im} = \underline{f}_{k_{im}, k_{im-1}, \dots, k_{(i-1)m+1}}(\underline{x}_{(i-1)m}).$$

Obviously, this is a first order recurrence with respect to i . Note that for $m=1$ the above general expression degenerates to the first order recurrence expression.

The conversion of the f_{k_i} -s into $f_{k'_i}$ -s is done by a node called zero-level-node (for it is located right before the first level of the tree-like network). The zero-level-node has $m-1$ parts, where part j evaluates $f_{k'_{(i-1)m+j}}$, for all j , $2 \leq j \leq m$. The reason for skipping $j=1$ is that $f_{k'_{(i-1)m+1}} = f_{k_{(i-1)m+1}}$, so this conversion is really identity (therefore the solution for a first order recurrence does not contain a zero-level-node).

We assume, without loss of generality, that m divides n . Then the solution of the recurrence system $R(m)$ of length n , denoted by $R\langle n, m \rangle$, is equal to the solution $R\langle n/m, 1 \rangle$ of the transformed system $R(1)$. In fact, the length of the transformed recurrence is reduced by a factor of m , but the tree-nodes of the transformed recurrence tend to be more complex, and unlike the first order recurrence, zero-level-nodes are introduced. Therefore, the solution of a high order recurrence is more complex, in general.

3. Network Structure for Serial Solution

The serial network utilizes n identical nodes, which will be called serial-nodes. Figure 1 shows the structure of a serial network that produces the solution $B\langle 4,1 \rangle$ of a recurrence system $B(1)$. \underline{x}_i and \underline{k}_i are Boolean vectors, in the general case, that is, $\underline{x}_i \in B_1 \times B_2 \times \dots \times B_t$, and $\underline{k}_i \in B_1 \times B_2 \times \dots \times B_s$ where $B_j = \{0,1\}$.

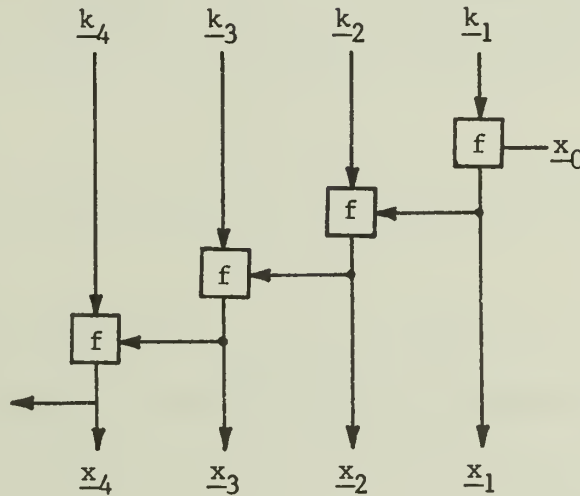


Figure 1. Serial network structure for $B\langle 4,1 \rangle$.

Adopting the terminology of finite state machines, the inputs from

the right side of each node define the present state. The next state is a function of the present state and the inputs entering the top of the node:

$$\underline{x}_i = f(\underline{k}_i, \underline{x}_{i-1}).$$

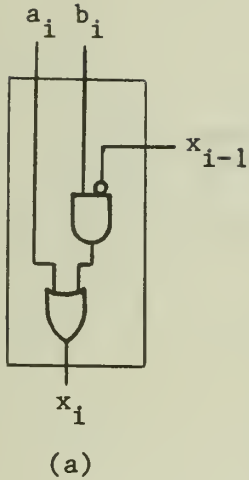
For every present state and input there exists the next state. We can look at the input \underline{k}_i as defining a mapping (called a function in section 2) $f_{\underline{k}_i}$ from the set of states X to the same set of states:

$$\underline{x}_i = f_{\underline{k}_i}(\underline{x}_{i-1}).$$

For $\underline{k}_i \neq \underline{k}_j$ we may have $f_{\underline{k}_i} = f_{\underline{k}_j}$. The total number of mappings is bounded. If the cardinality of \underline{k}_i is denoted by $C(\underline{k}_i)$, then there are at most $2^{C(\underline{k}_i)}$ mappings.

A serial-node for $B_2(1)$ (see section 2) and the three different mappings defined by the input bits a_i and b_i are shown in figure 2.

Figure 3 shows the structure of a serial network that produces the solution $B\langle 4, 2 \rangle$ of a second order recurrence system $B(2)$, for which $\underline{x}_i = f_{\underline{k}_i}(\underline{x}_{i-1}, \underline{x}_{i-2})$.



| | f_1 | f_2 | f_3 |
|---------------|-------|-------|--------|
| $a_i b_i =$ | 00 | 01 | 10, 11 |
| $x_{i-1} = 0$ | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 |

x_i

(b)

Figure 2. Recurrence system $B_2(1)$: (a) Serial-node in the serial network. (b) Mappings.

Example: $B_3(2) = \langle A \times B, X, (x_0, x_{-1}), F_3 \rangle$ and $F_3: x_i = a_i \bar{x}_{i-1} x_{i-2} + b_i x_{i-1}$. The serial-node in the serial network and the mappings are shown in figure 4.

Any Boolean recurrence of order $m > 1$ can be transformed into a first order Boolean recurrence, by grouping each m nodes together (using the idea presented section 2). Figure 5 shows the result of applying this transformation on the serial network that produces the solution $B\langle 4, 1 \rangle$,

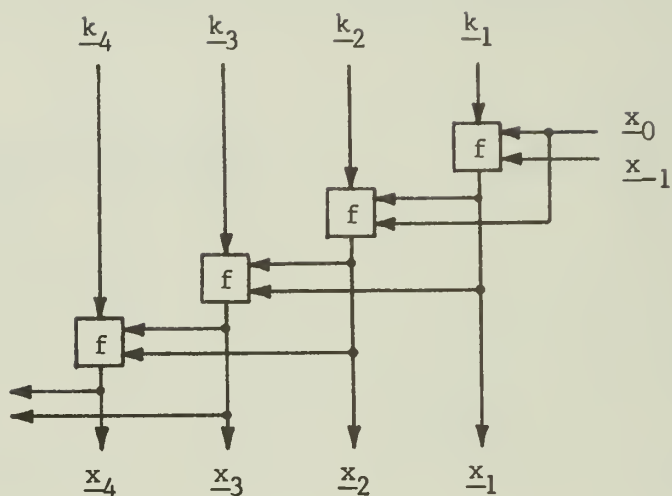


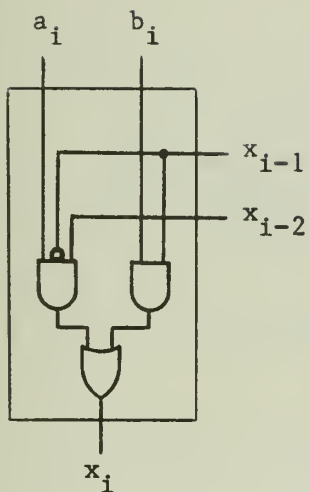
Figure 3. Serial network structure for $B\langle 4,2 \rangle$.

described in figure 3. The serial-node is surrounded with a dashed box. In fact, the resulting serial network can be viewed as producing the solution $B\langle 2,1 \rangle$.

Now

$$\underline{x}_{2i-1} = f_{\underline{k}_{2i-1}}(\underline{x}_{2i-2}, \underline{x}_{2i-3})$$

$$\underline{x}_{2i} = f_{\underline{k}_{2i}}(\underline{x}_{2i-1}, \underline{x}_{2i-2})$$



(a)

| | f_1 | f_2 | f_3 | f_4 |
|------------------------|-------|-------|-------|-------|
| $a_i b_i =$ | 00 | 01 | 10 | 11 |
| $x_{i-1} x_{i-2} = 00$ | 0 | 0 | 0 | 0 |
| 01 | 0 | 0 | 1 | 1 |
| 10 | 0 | 1 | 0 | 1 |
| 11 | 0 | 1 | 0 | 1 |

(b)

Figure 4. Recurrence system $B_3(2)$: (a) Serial-node in the serial network. (b) Mappings.

$$= f_{k_{2i}}(f_{k_{2i-1}}(x_{2i-2}, x_{2i-3}), x_{2i-2})$$

$$= f'_{k_{2i}, k_{2i-1}}(x_{2i-2}, x_{2i-3}).$$

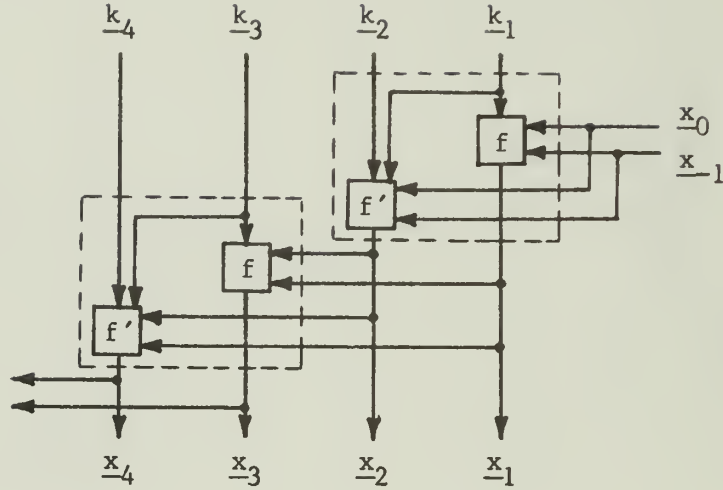


Figure 5. Serial network structure for $B\langle 4,2 \rangle$ transformed into serial network structure for $B\langle 2,1 \rangle$.

Each two states grouped together are evaluated in parallel, reducing the length of the recurrence by a factor of 2. However, it might not necessarily result in increasing the speed in general, because the part of the serial-node that evaluates f' tends to be more complex.

For example, transforming $B_3(2)$ into first order recurrence yields the following serial-node functions:

$$f: x_{2i-1} = a_{2i-1} \bar{x}_{2i-2} x_{2i-3} + b_{2i-1} x_{2i-2}$$

which is the same as in the original serial network, and

$$\begin{aligned} f': x_{2i} &= a_{2i} \bar{x}_{2i-1} x_{2i-2} + b_{2i} x_{2i-1} \\ &= a_{2i-1} b_{2i} \bar{x}_{2i-2} x_{2i-3} + (a_{2i} \bar{b}_{2i-1} + b_{2i} b_{2i-1}) x_{2i-2} \end{aligned}$$

which is somewhat more complex than f . The serial-node in the transformed network is shown in figure 6.

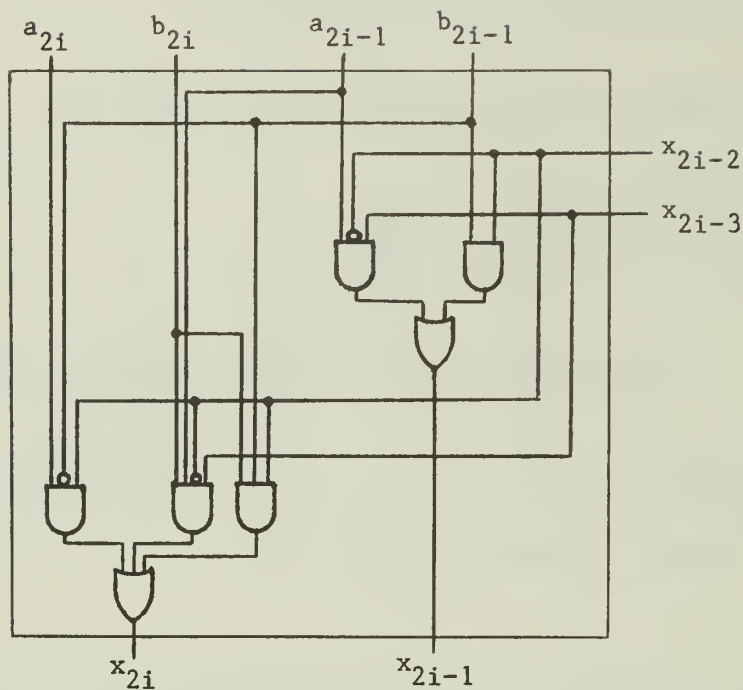


Figure 6. Transformed recurrence system $B_3(2)$: serial-node in the serial network.

4. Network Structure for Parallel Solution

Depending on the time when \underline{x}_0 is available and the time when the solution $\underline{x}_1, \dots, \underline{x}_n$ has to be stable, different networks can be constructed. Figure 7 shows the network that produces the solution $B\langle 8, 1 \rangle$ in the fastest way, assuming the pre-node, tree-node and out-node have the same delay d , and \underline{x}_0 is available $4d$ after the \underline{k}_i -s.

If the time constraints are released, such that the solution might be achieved in more time, or \underline{x}_0 is available earlier, a network with less tree-nodes can be constructed. However, the tree-nodes and the pre-nodes of this network are the same as those of the fastest network. Figure 8 shows an example of such a network, which will be called hybrid network, for it is a mixture of serial and parallel networks. In the extreme case, when \underline{x}_0 is available at the same time as the \underline{k}_i -s and the solution can be stable after sufficient long time, then the serial network (described in section 3) results.

Unlike the serial network, the number of different mappings here is not bounded by $2^{C(\underline{k}_i)}$. If the number of the Boolean equations describing F is b , then the number of states is bounded by $s = 2^b$ and the number of mappings is bounded by s^s . For example, F_2 in $B_2(1)$ is described by one equation, therefore $b=1$, $s=2$ and $s^s=4$.

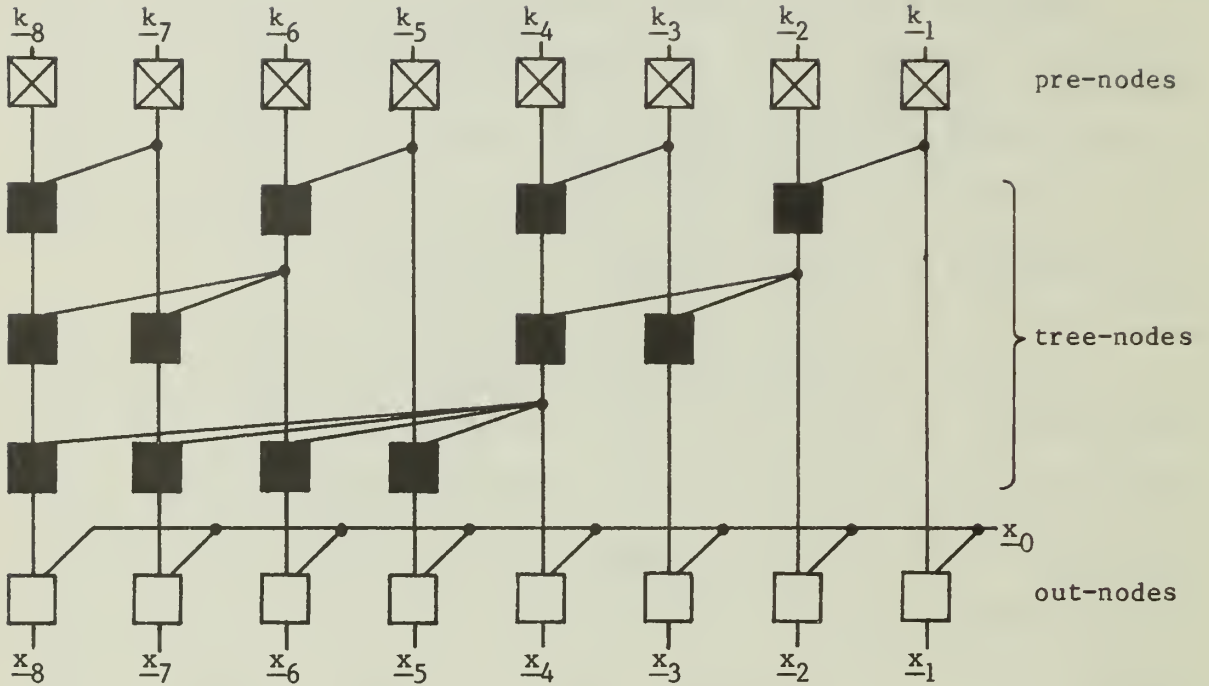


Figure 7. Parallel network structure for $B\langle 8,1 \rangle$.

As an example, figure 9 shows the tables and the nodes of the recurrence system $B_2(1)$.

The multiplication table (figure 9-b) is evaluated by composing each pair of mappings (see figure 2-b). This composition will be called multiplication from now on. From the mappings table, we have:

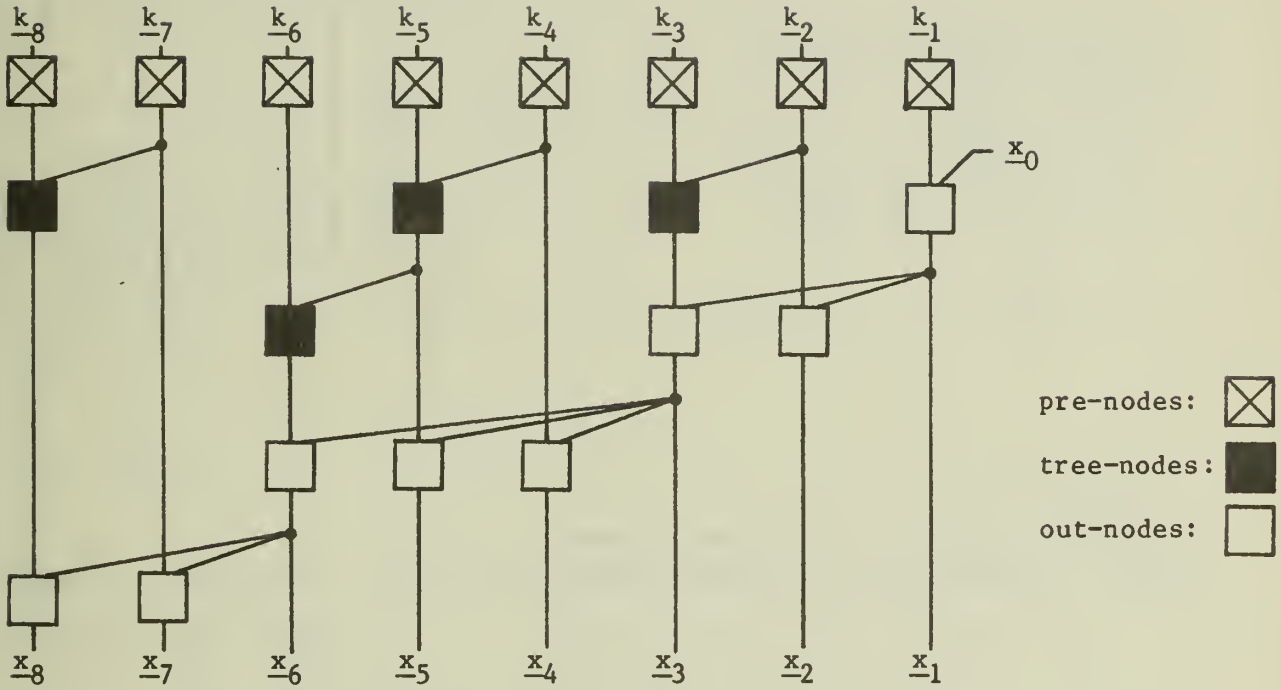


Figure 8. Hybrid network structure for $B\langle 8,1 \rangle$.

$$f_1: \begin{bmatrix} 0 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \text{or} \quad f_1 \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

$$f_2: \begin{bmatrix} 0 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad \text{or} \quad f_2 \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix},$$

$$f_3: \begin{bmatrix} 0 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad \text{or} \quad f_3 \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

| | f_1 | f_2 | f_3 | f_4 |
|-------------|-------|-------|-------|-------|
| $p_i q_i =$ | 00 | 10 | 11 | 01 |
| $x_0 = 0$ | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |

$$x_1 = p_1 \bar{x}_0 + q_1 x_0$$

(a)

| $f_j =$ | f_1 | f_4 | f_3 | f_2 |
|-------------|-------|-------|-------|-------|
| $f_i = f_1$ | f_1 | f_1 | f_1 | f_1 |
| f_4 | f_1 | f_4 | f_3 | f_2 |
| f_3 | f_3 | f_3 | f_3 | f_3 |
| f_2 | f_3 | f_2 | f_1 | f_4 |

$$f_k = f_i \circ f_j$$

(b)

| $f_{p_j q_j} =$ | f_{00} | f_{01} | f_{11} | f_{10} |
|------------------------|----------|----------|----------|----------|
| $f_{p_i q_i} = f_{00}$ | f_{00} | f_{00} | f_{00} | f_{00} |
| f_{01} | f_{00} | f_{01} | f_{11} | f_{10} |
| f_{11} | f_{11} | f_{11} | f_{11} | f_{11} |
| f_{10} | f_{11} | f_{10} | f_{00} | f_{01} |

$$f_{p_k q_k} = f_{p_i q_i} \circ f_{p_j q_j}$$

$$p_k = p_i \bar{p}_j + q_i p_j$$

$$q_k = p_i \bar{q}_j + q_i q_j$$

(c)

| $a_i b_i =$ | f_1 | f_2 | f_3 | f_3 |
|-------------|-------|-------|-------|-------|
| $p_i q_i =$ | 00 | 01 | 11 | 10 |
| | 00 | 10 | 11 | 11 |

$$p_i = a_i + b_i$$

$$q_i = a_i$$

(d)

Figure 9. Recurrence system $B_2(1)$: (a) Mappings table and out-node function. (b) Multiplication table. (c) Encoded multiplication table and tree-node function. (d) Conversion of input to mapping code and pre-node function. (continued on next page).

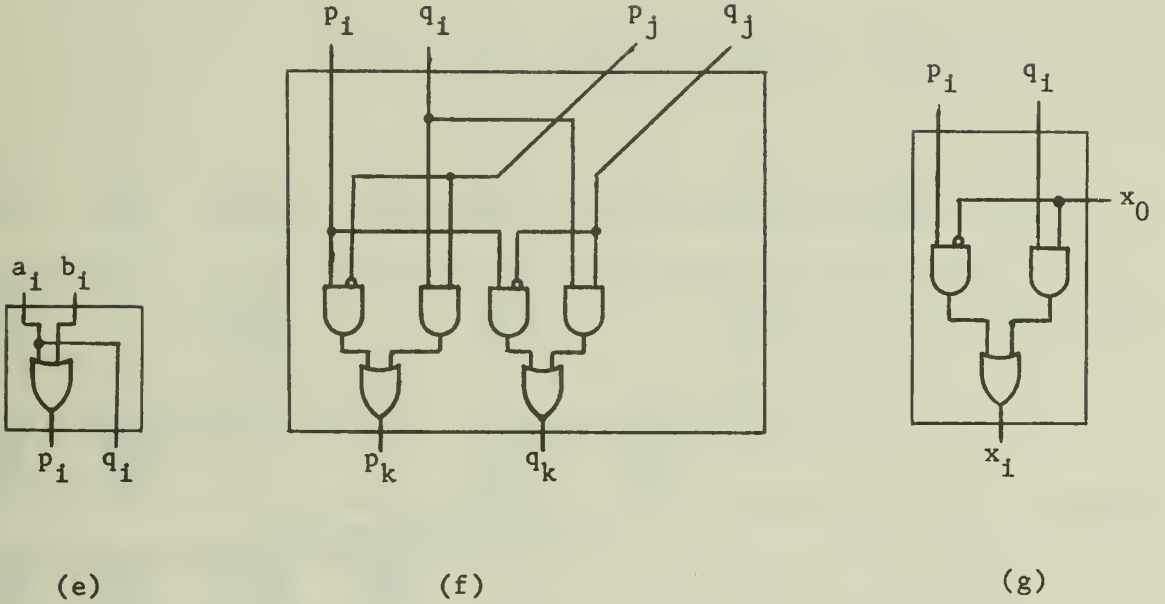


Figure 9 (continued). Recurrence system $B_2(1)$: (e) Pre-node. (f) Tree-node. (g) Out-node.

Thus, evaluation of $f_2^{\text{of}} f_3$, for example, is done in the following way:

$$(f_2^{\text{of}} f_3) \begin{bmatrix} 0 \\ 1 \end{bmatrix} = f_2(f_3 \begin{bmatrix} 0 \\ 1 \end{bmatrix}) = f_2 \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} = f_1 \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Therefore, $f_2^{\text{of}} f_3 = f_1$.

A new mapping, denoted by f_4 is generated when f_2 is multiplied by itself. Thus,

$$(f_2 \circ f_2) \begin{bmatrix} 0 \\ 1 \end{bmatrix} = f_2(f_2 \begin{bmatrix} 0 \\ 1 \end{bmatrix}) = f_2 \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = f_4 \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

f_4 is the identity mapping. The semigroup now contains four mappings, which is the maximum number of mappings for $B_2(1)$, as evaluated earlier in this section.

The next states $(f_1(0), f_1(1))$ under the mapping f_1 are used as the mapping code for mapping f_1 , i.e., mapping f_1 is assigned the code 00, f_2 : 10, f_3 : 11 and f_4 : 01 (p_1q_1 in figure 9-a). The encoded multiplication table is shown in figure 9-c. After the code assignment is determined, the pre-node is defined (figure 9-d).

The Boolean equations were derived by using the tables as Karnaugh maps.

For recurrence of order $m > 1$, the parallel network contains generally 4 different types of nodes: pre-nodes, tree-nodes and out-nodes as in the first order recurrence network, and a zero-level-node (see section 2). An example of a parallel network for a second order recurrence is shown in figure 10.

The network in figure 10 can be viewed as a network that produces the solution $B\langle 4, 1 \rangle$ for a first order recurrence system $B(1)$. Essentially the same methods for solving the first order recurrence apply

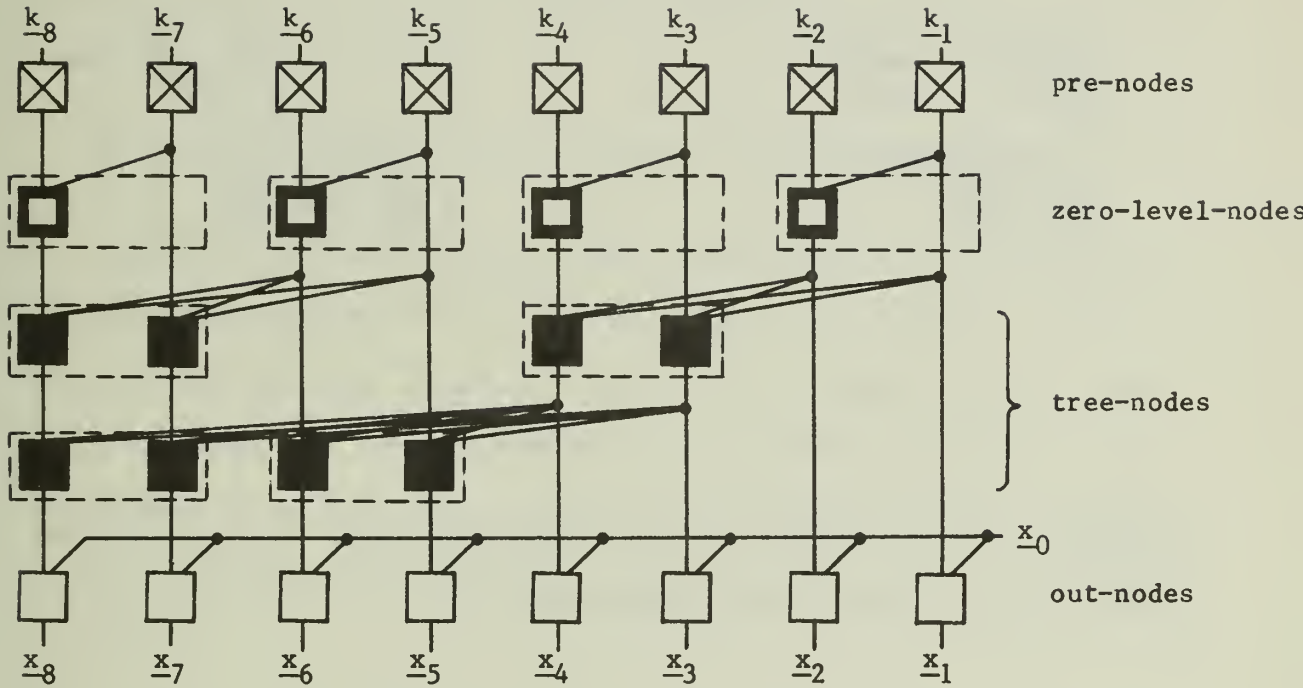


Figure 10. Parallel network structure for $B\langle 8,2 \rangle$.

here.

In general, the zero-level-node for Boolean recurrence of order m can be constructed in two ways, as shown in figure 11, for $m=4$.

The nodes in figure 11-a are the tree-nodes with some constant



Figure 11. Zero-level-node for $B(4)$: (a) Constructed from degenerated tree-nodes. (b) Constructed independently.

inputs in their disconnected lines. Therefore their Boolean functions can be simplified.

The nodes in figure 11-b are different from each other, and are called part 1, part 2 and part 3 of the zero-level-node (in general there are $m-1$ such parts). Their Boolean functions are defined by the conversion of the codes for the f_{k_1} -s into the codes for the $f_{k'_1}$ -s, as outlined in section 2.

The recurrence system $B_3(2)$ is used as an example. The various

tables and nodes are shown in figure 12. The mappings f_1 , f_2 , f_3 and f_4 are assigned the codes 00, 01, 10 and 11 respectively, the same as the input (p_1q_1) in figure 12-a).

The generation of the tables was done in the following order:

1. Zero-level conversion table (figure 12-b). Each pair of two possible input mappings (f_i, f_j) (total of 16) were multiplied to form a two-mappings group $(f_i \circ f_j, f_j)$, which is the output of the zero-level-node. No new mapping has been generated in this step. The result is 10 different groups, each containing two mappings: f_1, f_1 ; f_2, f_1 ; f_1, f_2 ; f_2, f_2 ; f_1, f_3 ; f_3, f_3 ; f_2, f_3 ; f_4, f_3 ; f_1, f_4 ; f_4, f_4 .
2. Multiplication table (figure 12-c). Each pair of groups (f_i, f_j) , (f_p, f_q) (not necessarily distinctive) generated in step 1 were composed to form a group $((f_i, f_j) \circ (f_p, f_q))$. From now on, composition of groups will be called g-multiplication, to distinguish from composition of mappings, which is called simply multiplication. Composition of a mapping and a group will be called m-multiplication. Two additional groups have been created: f_3, f_1 and f_4, f_1 . Each of the additional groups was g-multiplied by the other groups the same way as the first 10 groups, but no new group has been generated. Also, no new mapping has been generated in this step.

In this step, groups are g-multiplied by groups, but it is sufficient to hold in the mappings table only the result of the multiplication of a mapping by a group. Thus, evaluation of group g-multiplication from the table is done by using distributivity:

$$(f_i, f_j) \circ (f_p, f_q) = f_i \circ (f_p, f_q), f_j \circ (f_p, f_q).$$

For example, $(f_2, f_3) \circ (f_4, f_3) = f_2 \circ (f_4, f_3), f_3 \circ (f_4, f_3) = f_4, f_1$.

The evaluation of the nodes' Boolean equations was done after the tables were encoded, by using Quine-McCluskey minimization technique [3].

Generally, if a new mapping is generated during step 1 or step 2, it is appended to the mappings table. In the example above, no new mapping was generated in either step, so the original mappings table has not been changed and still contains only the original four mappings. This fact and the assignment of the input bits as the mappings codes result in a pre-node which is the identity, i.e., $e_i = a_i$ and $f_i = b_i$, so in fact, in this case there exists no pre-node at all.

| | f_1 | f_2 | f_3 | f_4 |
|-------------------|-------|-------|-------|-------|
| $p_i q_i =$ | 00 | 01 | 10 | 11 |
| $x_0 x_{-1} = 00$ | 0 | 0 | 0 | 0 |
| 01 | 0 | 0 | 1 | 1 |
| 10 | 0 | 1 | 0 | 1 |
| 11 | 0 | 1 | 0 | 1 |

$$x_i = p_i \bar{x}_0 x_{-1} + q_i x_0$$

(a)

| | |
|---------------------------------|---------------------------------|
| $f_1, f_1 \rightarrow f_1, f_1$ | $f_1, f_3 \rightarrow f_1, f_3$ |
| $f_2, f_1 \rightarrow f_1, f_1$ | $f_2, f_3 \rightarrow f_3, f_3$ |
| $f_3, f_1 \rightarrow f_2, f_1$ | $f_3, f_3 \rightarrow f_2, f_3$ |
| $f_4, f_1 \rightarrow f_2, f_1$ | $f_4, f_3 \rightarrow f_4, f_3$ |
| $f_1, f_2 \rightarrow f_1, f_2$ | $f_1, f_4 \rightarrow f_1, f_4$ |
| $f_2, f_2 \rightarrow f_2, f_2$ | $f_2, f_4 \rightarrow f_4, f_4$ |
| $f_3, f_2 \rightarrow f_1, f_2$ | $f_3, f_4 \rightarrow f_1, f_4$ |
| $f_4, f_2 \rightarrow f_2, f_2$ | $f_4, f_4 \rightarrow f_4, f_4$ |

$$p_{2i} = e_{2i-1} f_{2i}$$

$$q_{2i} = e_{2i} \bar{f}_{2i-1} + f_{2i} f_{2i-1}$$

(b)

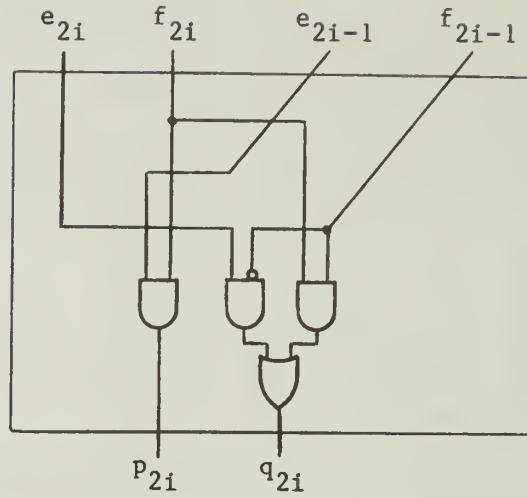
| | f_1, f_1 | f_2, f_1 | f_1, f_2 | f_2, f_2 | f_1, f_3 | f_3, f_3 | f_2, f_3 | f_4, f_3 | f_1, f_4 | f_4, f_4 | f_3, f_1 | f_4, f_1 |
|-------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|
| f_1 | f_1 | f_1 | f_1 | f_1 | f_1 | f_1 | f_1 | f_1 | f_1 | f_1 | f_1 | f_1 |
| f_2 | f_1 | f_2 | f_1 | f_2 | f_1 | f_3 | f_2 | f_4 | f_1 | f_4 | f_3 | f_4 |
| f_3 | f_1 | f_1 | f_2 | f_1 | f_3 | f_1 | f_3 | f_1 | f_4 | f_1 | f_1 | f_1 |
| f_4 | f_1 | f_2 | f_2 | f_2 | f_3 | f_3 | f_4 | f_4 | f_4 | f_4 | f_3 | f_4 |

$$p_k = p_i \bar{p}_{2j} p_{2j-1} + q_i p_{2j}$$

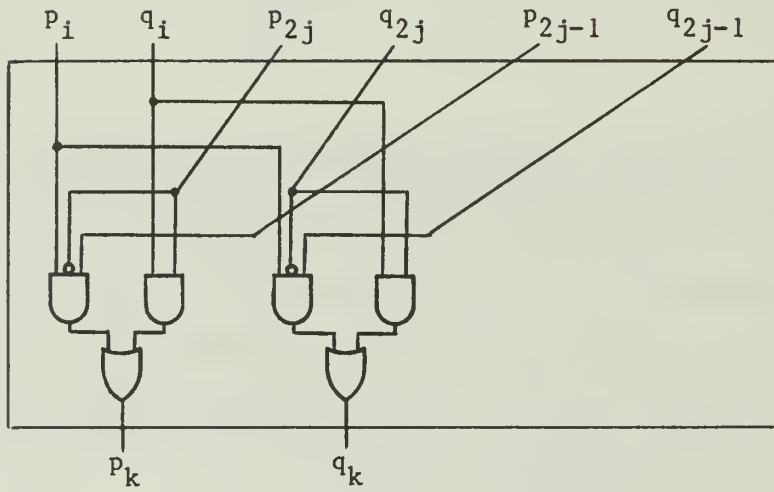
$$q_k = p_i \bar{q}_{2j} q_{2j-1} + q_i q_{2j}$$

(c)

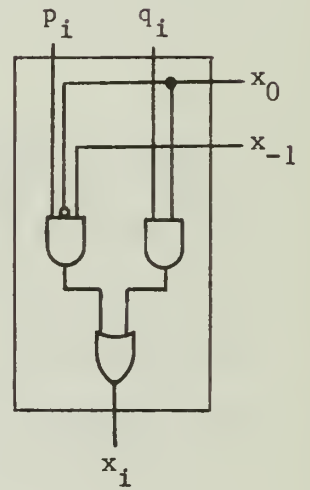
Figure 12. Recurrence system $B_3(2)$: (a) Mappings table and out-node function. (b) Zero-level conversion table and zero-level-node function. (c) Multiplication table and tree-node function. (continued on next page).



(d)



(e)



(f)

Figure 12 (continued). Recurrence system $B_3(2)$: (d) Zero-level-node. (e) Tree-node. (f) Out-node.

5. The Programs and their Role

Three programs participate in the nodes' equations generation: PREPRC, BINREC, and MINIMZ. Their sequence of running and the I/O files for each program are shown in figure 13.

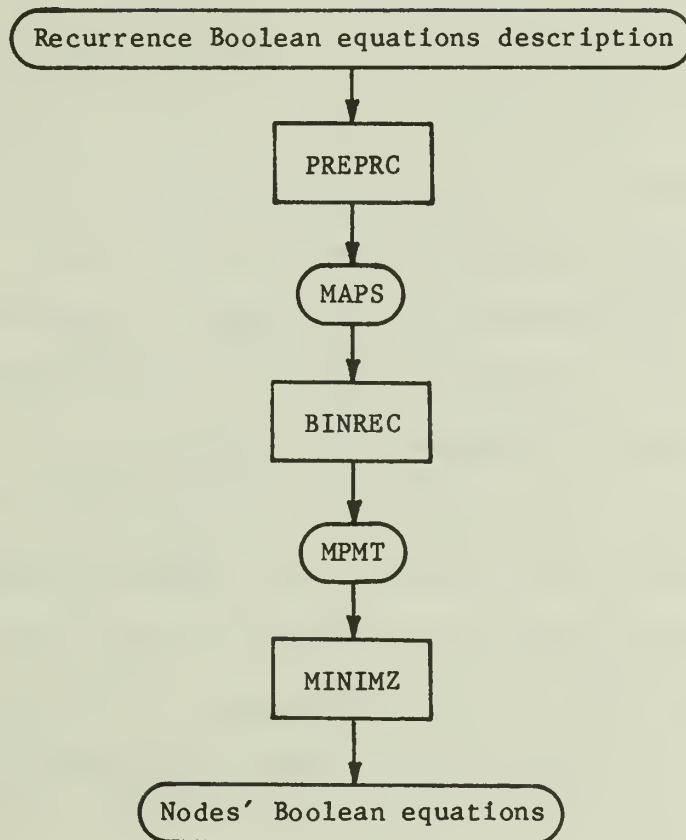


Figure 13. Programs' sequence.

5.1 PREPRC

This program preprocesses the Boolean equations description. Its main input is the description of the Boolean equations in a table form, which will be clear from the examples. Additional inputs are the order of the recurrence, the number of Boolean equations in the recurrence function F , the cardinality of \underline{k}_1 and the starting state \underline{x}_0 .

Let the input bits be given each a binary value, independently. Then the certain image of the input bits is called a pattern. The output of the program is the file MAPS, of which the main part is the initial mappings table, which defines the next state for every given present state and input pattern. If the number of Boolean equations is b , then the number of states is at most 2^b . If the number of input bits is k , then the number of different input patterns is at most 2^k . However, two or more input patterns can define the same mapping, so the number of mappings is less or equal to the number of different input patterns. The input patterns corresponding to a particular mapping are called the natural assignment of that mapping. the natural assignments are also part of the program output. Additional outputs are the states' codes.

Figure 14 shows the input to PREPRC and the initial mappings table for recurrence system $B_2(1)$.

| | $\bar{a}_i \bar{b}_i$ | $\bar{a}_i b_i$ | $a_i \bar{b}_i$ | $a_i b_i$ |
|-----------------|-----------------------|-----------------|-----------------|-----------|
| 1 | 0 | 0 | 1 | 1 |
| \bar{x}_{i-1} | 0 | 1 | 0 | 1 |
| x_{i-1} | 0 | 0 | 0 | 0 |

$$x_i = a_i + b_i \bar{x}_{i-1}$$

(a)

| | | | | | | |
|-------------|---|---|----|----|----|----------------------|
| | | | 00 | 01 | 1- | ← natural assignment |
| | | | 1 | 2 | 3 | ← mapping # |
| $x_{i-1} =$ | 0 | 1 | 1 | 2 | 2 | |
| | 1 | 2 | 1 | 1 | 2 | |

↑
present state #

(b)

Figure 14. Recurrence system $B_2(1)$: (a) Input table. (b) Initial mappings table.

Notes: (1) The state # and the mapping # are shown here only for clearness.

(2) The dash (-) denotes a don't care bit (0 or 1).

In certain cases, assuming a specified initial state, there are states that are unreachable. In order to get a simpler initial mappings table, those states have to be eliminated. Example for such a case is the comparator. Its recurrence system is $B_4(1) = \langle A \times B, X \times Y, (x_0, y_0), F_4 \rangle$, with

$$F_4: x_i = x_{i-1} + a_i \bar{b}_i \bar{y}_{i-1},$$

$$y_i = y_{i-1} + \bar{a}_i b_i \bar{x}_{i-1}.$$

Figure 15-a shows the input table. Figure 15-b shows the original initial mappings table, implied by the equations of F_4 .

For the comparator to function properly, it is necessary to assume that the initial state is $x_0 y_0 = 00$. States 00, 01 and 10 can be reached from state 00, while 11 cannot be reached from any state, other than itself. Therefore, state 11 is eliminated and we end up with a simpler initial mappings table, shown in figure 15-c.

As a final example, we will consider the second order Boolean recurrence system $B_3(2)$, whose input table and the resulting initial mappings table are shown in figure 16.

| | | x_i | | | | y_i | | | |
|-------------------------------|--|-----------------------|-----------------|-----------------|-----------|-----------------------|-----------------|-----------------|-----------|
| | | $\bar{a}_i \bar{b}_i$ | $\bar{a}_i b_i$ | $a_i \bar{b}_i$ | $a_i b_i$ | $\bar{a}_i \bar{b}_i$ | $\bar{a}_i b_i$ | $a_i \bar{b}_i$ | $a_i b_i$ |
| 1 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| \bar{x}_{i-1} | | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| x_{i-1} | | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| \bar{y}_{i-1} | | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| y_{i-1} | | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| $\bar{x}_{i-1} \bar{y}_{i-1}$ | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $\bar{x}_{i-1} y_{i-1}$ | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $x_{i-1} \bar{y}_{i-1}$ | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $x_{i-1} y_{i-1}$ | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

$$x_i = x_{i-1} + a_i \bar{b}_i \bar{y}_{i-1}$$

$$y_i = y_{i-1} + \bar{a}_i b_i \bar{x}_{i-1}$$

(a)

Figure 15. Recurrence system $B_4(1)$: (a) Input table. (continued on next page).

| $a_i b_i =$ | 00 | 01 | 10 | 11 |
|------------------------|----|----|----|----|
| $x_{i-1} y_{i-1} = 00$ | 00 | 01 | 10 | 00 |
| 01 | 01 | 01 | 01 | 01 |
| 10 | 10 | 10 | 10 | 10 |
| 11 | 11 | 11 | 11 | 11 |

(b)

| | | 00, 11 | 01 | 10 | ← natural assignment |
|---------------------|----|--------|----|----|----------------------|
| | | 1 | 2 | 3 | ← mapping # |
| $x_{i-1} y_{i-1} =$ | 00 | 1 | 1 | 2 | 3 |
| | 01 | 2 | 2 | 2 | 2 |
| | 10 | 3 | 3 | 3 | 3 |

↑
 present state #

(c)

Figure 15 (continued). Recurrence system $B_4(1)$: (b) Initial mappings table before eliminating state 11. (c) Initial mappings table.

| | $\bar{a}_i \bar{b}_i$ | $\bar{a}_i b_i$ | $a_i \bar{b}_i$ | $a_i b_i$ |
|-------------------------------|-----------------------|-----------------|-----------------|-----------|
| 1 | 0 | 0 | 0 | 0 |
| \bar{x}_{i-1} | 0 | 0 | 0 | 0 |
| x_{i-1} | 0 | 1 | 0 | 1 |
| \bar{x}_{i-2} | 0 | 0 | 0 | 0 |
| x_{i-2} | 0 | 0 | 0 | 0 |
| $\bar{x}_{i-1} \bar{x}_{i-2}$ | 0 | 0 | 0 | 0 |
| $\bar{x}_{i-1} x_{i-2}$ | 0 | 0 | 1 | 1 |
| $x_{i-1} \bar{x}_{i-2}$ | 0 | 0 | 0 | 0 |
| $x_{i-1} x_{i-2}$ | 0 | 0 | 0 | 0 |

$$x_i = a_i \bar{x}_{i-1} x_{i-2} + b_i x_{i-1}$$

(a)

Figure 16. Recurrence system $B_3(2)$: (a) Input table. (continued on next page).

| | | | 00 | 01 | 10 | 11 | ← natural assignment |
|--------------------|----|-----|----|----|----|----|----------------------|
| | | | 1 | 2 | 3 | 4 | ← mapping # |
| $x_{1-1}x_{1-2} =$ | 00 | 1,1 | 1 | 1 | 1 | 1 | |
| | 01 | 1,2 | 1 | 1 | 2 | 2 | |
| | 10 | 2,1 | 1 | 2 | 1 | 2 | |
| | 11 | 2,2 | 1 | 2 | 1 | 2 | |

↑
previous and present state #'s

(b)

Figure 16 (continued). Recurrence system $B_3(2)$: (b) Initial mappings table.

5.2 BINREC

This program receives the file MAPS generated by the program PREPRC and produces the file MPMT, of which the main part is the mappings table and the multiplication table, and for recurrences of order $m > 1$, also the zero-level conversion table. Additional outputs are the mappings' natural assignments and the states' codes, as passed in the file MAPS.

As shown in section 4, the zero-level conversion table defines the zero-level-node, the mappings table defines the out-node and the multiplication table defines the tree-node. The algorithm that evaluates the tables has two parts. The first part evaluates the zero-level conversion table, then the second part evaluates simultaneously the two other tables.

Before the presentation of the algorithm, we will introduce the data structures for the tables. The tables, after the execution of the algorithm on the recurrence system $B_3(2)$ (refer to figure 12), are shown in figure 17.

The representation of the mappings table (two-dimensional array MP) in figure 17-a is self explanatory. There is no need for columns or rows headings, since the mappings numbers and the states numbers form the indices to this array. Column j corresponds to mapping f_j , while the

row is computed in somewhat different manner. If we look at the pairs of states as an ordered set, then pair 1,1 is related to row 1, pair 1,2 to row 2, 2,1 to 3 and 2,2 to 4. In the case of a first order recurrence, the situation is degenerated to a simple order, that is, row i corresponds to state number i .

The representation of the multiplication table (two-dimensional array MT) is shown in figure 17-b. Here there is no need for rows headings, since row i corresponds to mapping f_i . The columns headings are the numbers of the mappings that form a group.

The representation of the zero-level conversion table (one-dimensional array ZERO) is shown in figure 17-c. The zero-level conversion table contains pointers to the columns headings of the multiplication table. The index to the table is a simple function of the numbers of the mappings in the input group of mappings. In this example, the number of mappings in the initial mappings table is $LI=4$. If we want to know to which group of mappings is f_i, f_j converted, we look at the entry with the index $r = (j-1)*LI + (i-1) + 1$. The contents $ZERO[r]$ are the index of the mappings group in array MT. For example, f_3, f_2 is converted to f_1, f_2 , since $r = (2-1)*4 + (3-1) + 1 = 7$, $ZERO[7]=3$, and the mappings group in the heading of column 3 of MT is f_1, f_2 . In the case of a first order recurrence, this array is not needed (it degenerates to the

| | | | |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 1 | 1 | 2 | 2 |
| 1 | 2 | 1 | 2 |
| 1 | 2 | 1 | 2 |

MP

(a)

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 1 | 2 | 1 | 3 | 2 | 4 | 1 | 4 | 3 | 4 |
| 1 | 1 | 2 | 2 | 3 | 3 | 3 | 3 | 4 | 4 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 2 | 1 | 2 | 1 | 3 | 2 | 4 | 1 | 4 | 3 | 4 |
| 1 | 1 | 2 | 1 | 3 | 1 | 3 | 1 | 4 | 1 | 1 | 1 |
| 1 | 2 | 2 | 2 | 3 | 3 | 4 | 4 | 4 | 4 | 3 | 4 |

MT

(b)

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|----|---|----|
| 1 | 1 | 2 | 2 | 3 | 4 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|----|---|----|

ZERO

(c)

Figure 17. Data structures for recurrence system $B_3(2)$: (a) Array MP: mappings table. (b) Array MT: multiplication table. (c) Array ZERO: zero-level conversion table.

ordered set of mappings numbers, that is, 1,2,...).

The indexing shown for the three arrays can be easily generalized to higher order recurrences.

The arrays hold the information needed with no redundancy, therefore the utilized memory space is minimal, yet the access to the data does not require too much manipulation.

Part (a) of the algorithm

This part evaluates array ZERO and some of the columns headings of array MT.

For a recurrence of order m , each group of m mappings $f_{i_m}, \dots, f_{i_2}, f_{i_1}$ is converted by the zero-level-node to $f_{i'_m}, \dots, f_{i'_2}, f_{i'_1}$ where

$$f_{i'_1} = f_{i_1}$$

$$f_{i'_2} = f_{i_2} \circ f_{i_1}$$

.

.

.

$$f_{i'_m} = f_{i_m} \circ \dots \circ f_{i_2} \circ f_{i_1}.$$

If the number of mappings in the mappings table is LI, then each index i_j takes the values 1 to LI, therefore LI^m different groups are converted.

Every $f_{i'_m}, \dots, f_{i'_2}, f_{i'_1}$ group is appended to MT, if it is not already there, and the corresponding entry in ZERO is evaluated. If new mappings are generated during this part of the algorithm (one or few among the mappings $f_{i'_2}, \dots, f_{i'_m}$), they are appended to MP.

Note: Part (a) of the algorithm might be skipped for first order recurrence, since always $f_{i'_1} = f_{i_1}$.

Part (b) of the algorithm

This part completes the evaluation of arrays MP and MT. The evaluation is done in a finite number of passes over the groups of mappings that are stored in the columns headings of MT.

On the first pass, every pair of mappings groups is g-multiplied (including g-multiplication of a group by itself), and the corresponding entries in MT are evaluated, if not already evaluated in a former g-multiplication. If, for example, in the second order recurrence we compute $f_u, f_v = (f_i, f_j) \circ (f_p, f_q)$ then in the column corresponding to the mappings group f_p, f_q , u is stored in row i and v is stored in row j, if not stored there yet.

If a new mappings group is generated, it is appended to MT. If new mappings are generated, they are appended to MP.

If new mappings groups were generated in the first pass, a second pass is performed, in which only pairs that were not g-multiplied in the first pass are taken into account.

Additional passes are performed if necessary, until no new mappings group is generated. Since the semigroup is finite, the algorithm is guaranteed to terminate.

Figure 18 shows some typical snapshots of arrays MP and MT during the execution of part (b) of the algorithm. Shaded areas show the evaluated entries.

Note: Each pass corresponds to a g-multiplication of mappings groups at some different level of the tree-like network. i.e., first pass - to level 1 (right below the zero-level), second pass - to level 2, and so on. If there are more levels than passes, then the last pass corresponds to all of the remaining levels. Since the number of mappings groups is different at each level, a different type of node can be constructed for each level. However, for the sake of uniformity (which is a major consideration in VLSI design), one type of node is designed, which suits all of the mappings groups.

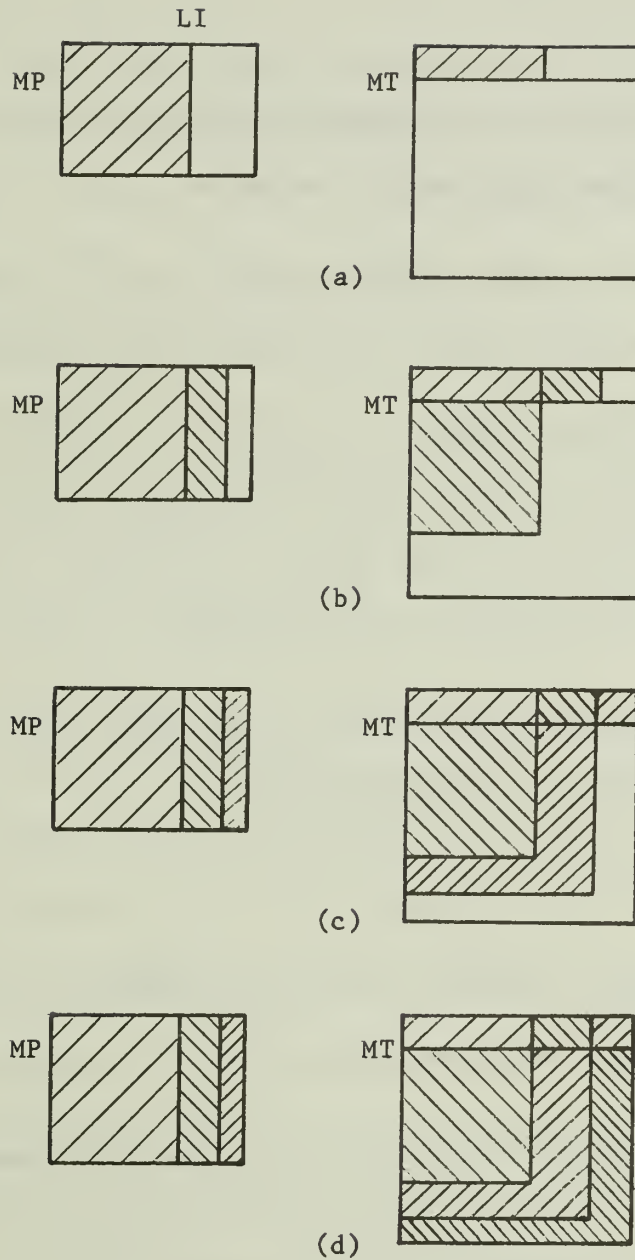


Figure 18. Development of MP and MT during execution of part (b) of the algorithm: (a) After part (a), just before part (b). (b) After first pass. (c) After second pass. (d) After third (in this case last) part.

5.3 MINIMZ

This program receives the file MPMT generated by the program BINREC and computes the Boolean equations of the nodes.

The complexity of these Boolean equations heavily depends on the encoding of each mapping, making code assignment very important.

Since checking every possible assignment is not practical, some heuristic approaches are taken in order to get the (near) optimal solution. In fact, for some well known Boolean recurrences (such as the adder, comparator and others) the optimal solution is achieved.

The code assignment is done in one or two modes, and in one, two or three sub-modes, as will be explained below.

The two modes are:

- (a) Natural assignment. Each of the initial mappings is assigned its natural assignment code (see section 5.1).

If a mapping has more than one code of natural assignment (see, for example, figure 15-c), then only the first one is used. However, if several natural assignment codes are combined through don't cares (see, for example, figure 14-b), then the code with the don't care is assigned as it is. Then attempt is made to reduce the number of

bits in the code (this can be done if 2^b is greater or equal to twice the number of the initial mappings, where b is the number of the bits in the code). If the mappings table contains only the original initial mappings, that is, no new mappings have been added, then we are done. If new mappings have been added, they are assigned codes by performing the following steps:

1. If the number of mappings in the mappings table is such that the code must be expanded, then the codes already assigned to the initial mappings are augmented with sufficient number of bits, which are initially set to the don't care value.
2. If an initial mapping has natural assignment codes which are not assigned to that mapping (recall that only the first code is assigned), they are assigned to the additional mappings in the mappings table. If every mapping is now assigned a code - we are done, otherwise next step must be carried out.
3. If a mapping has a code which contains at least one don't care bit (which must be true if this step is reached), its code can be split into two codes: one has 0 instead of the don't care and the other has 1. The other bits are left unchanged. One code is assigned to the mapping which originally was assigned the splitted code, and the other - to a mapping that has not

been assigned any code yet. Generally, there is no preference as to which mapping is assigned the code with the don't care changed to 0 and which mapping is assigned the other. However, in the case in which the Boolean recurrence function is defined by just one equation, some improvement can be achieved by making the code similar to the mapping. This is done the following way. First, the states that construct the mappings are regarded as having the values 0 and 1 instead of 1 and 2, respectively. Then, if the don't care is, say, in position r of the code, it is replaced by the value of the state in position r of the mapping. This heuristic gives better results in some cases. The mappings table is scanned from right to left, splitting the codes that contain the don't cares, until all the mappings have been assigned codes. Sometimes one scan or less is enough, but sometimes more than one scan is needed, generally if some codes contain more than one don't care each. The reason for scanning from right to left is that this way more don't cares are left in the codes of the initial mappings, resulting in a simpler pre-node.

(b) Mappings assignment. This assignment is applicable only in the case in which the Boolean recurrence function is defined by just one equation. In this case, the states 1 and 2 are regarded as 0 and 1, respectively, then each mapping's code is taken as the sequence of the states in the mapping. Since the mappings are different from each other, the code is guaranteed to be legal (that is, no more than one mapping is assigned the same code). Then attempt is made, as in the natural assignment mode, to reduce the number of bits in the code.

Now we have a lot of freedom, the amount of which depends on the number of mappings and the number of bits in the code: we can leave the codes as they are, or change some of the zeroes or some of the ones or some combination of zeroes and ones into don't cares. Among all these possibilities, the three sub-modes have been chosen, as follows:

- (a) No change of ones and zeroes into don't cares, that is, the code is left untouched.
- (b) Changing as many zeroes as possible into don't cares, then as many ones as possible into don't cares.

(c) Same as (b), except first the ones are changed, then the zeroes.

Sub-mode (a) can always be accomplished, but sometimes sub-mode (b) or (c) or both cannot.

After the assignment is determined, the Boolean equations of the nodes are evaluated, in the sum of products form, using Quine-McCluskey method for minimization [3].

6. Examples

The examples show the solutions of the following Boolean recurrences systems: $B_2(1)$, $B_4(1)$ and $B_3(2)$. The input format is as follows:

- (a) Comment line.
- (b) Recurrence parameters: order, number of equations, number of input bits (represented by decimal numbers) and the initial state code (in the form of separated binary digits), in one line.
- (c) Table(s) describing the Boolean equation(s). The right portion of these lines can be used for comments.

The main output is the Boolean equations of the nodes. X' denotes the complement of X . Example of the notation is shown in figure 19, for the four types of nodes, assuming third order recurrence with two bits for mapping code and one bit for state code. This notation can be easily generalized for any order recurrence and any number of bits for mapping and state codes.

Notes: (1) For the first order Boolean recurrence, the zero-level-node does not exist, therefore the pre-node has a different notation for its outputs.

(2) The term bunch is equivalent to group of mappings in the text.

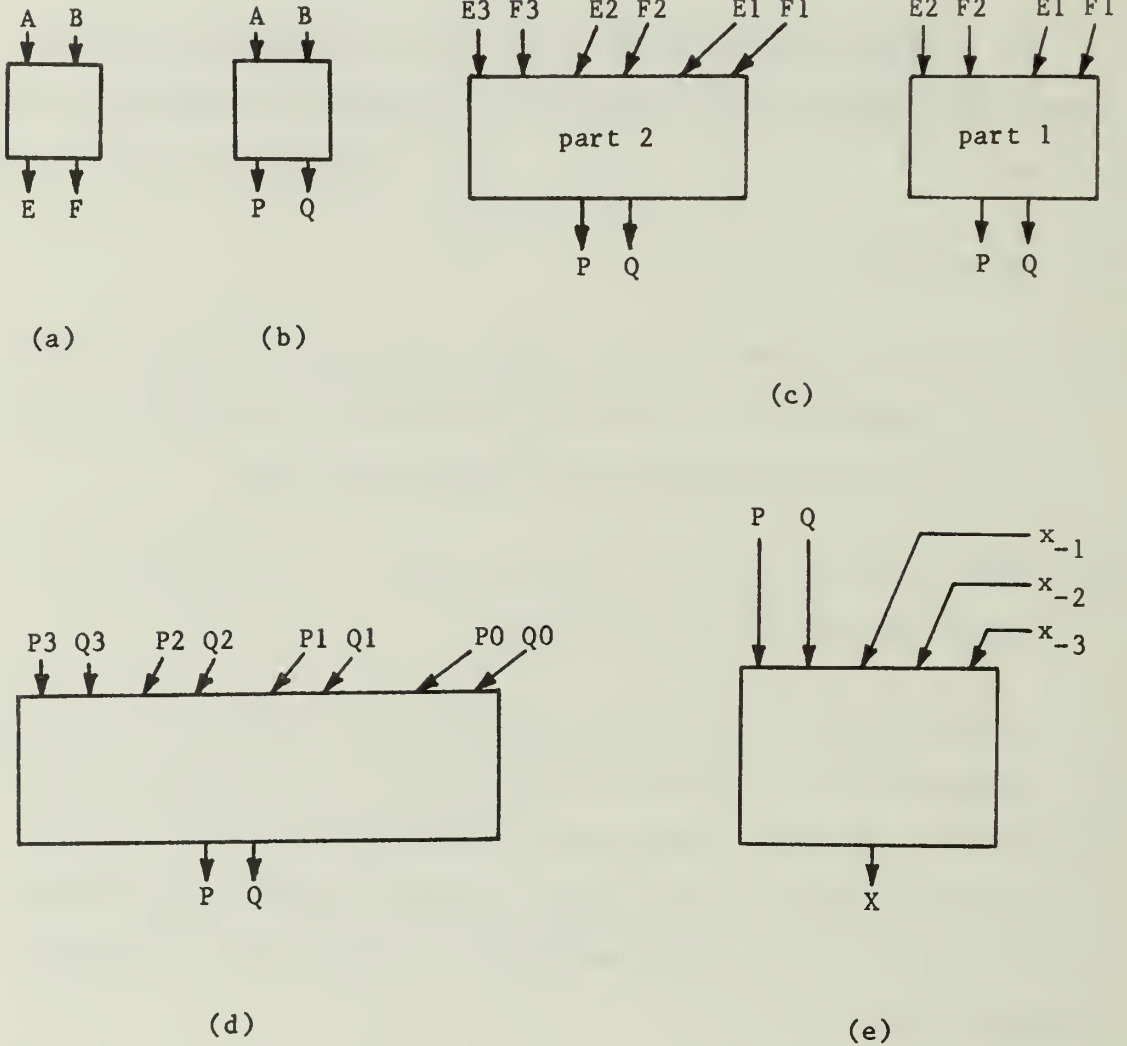


Figure 19. Notation of the input and output lines: (a) Pre-node for Boolean recurrence of order > 1 . (b) Pre-node for first order Boolean recurrence. (c) Zero-level-node. (d) Tree-node. (e) Out-node.

```

X(I) = A(I) + B(I)*X(I-1),
1 1 2 0
0 0 1 1
0 1 0 1
0 0 0 0

```

[illegible]

```

SECOND ORDER EXAMPLE.  X(I) = A(I)*X(I-1)'*X(I-2) + B(I)*X(I-1)
2 1 2 0
X(I) =
0 0 0 0
0 0 0 0
0 1 0 1
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 1 1
0 0 0 0
0 0 0 0
+ A(I)*X(I-1)'*X(I-2)

```

Input to PREPRC

$$B_2(1)$$
$$B_4(1)$$
 $B_3(2)$

X(I) = A(I) + B(I)*X(I-1)'

MAPPINGS TABLE:

| | ! | 1 | 2 | 3 | ! | 4 |
|---|------|---|---|---|------|---|
| | ---- | | | | ---- | |
| 1 | ! | 1 | 2 | 2 | ! | 1 |
| 2 | ! | 1 | 1 | 2 | ! | 2 |

STATES' CODES:

1: 0
2: 1

MAPPINGS' NATURAL ASSIGNMENTS:

1: 0 0
2: 0 1
3: 1 -

MULTIPLICATION TABLE:
(4 BUNCHES)

| | 4 | ! | 1 | 2 | 3 | ! | 4 |
|---|------|---|---|---|------|---|---|
| 4 | ! | 1 | 2 | 3 | ! | 3 | 4 |
| 3 | ! | 1 | 1 | 3 | ! | 3 | 3 |
| 2 | ! | 1 | 4 | 3 | ! | 2 | 2 |
| 1 | ! | 1 | 3 | 3 | ! | 1 | 1 |
| | ---- | | | | ---- | | |
| | ! | 1 | 2 | 3 | ! | 3 | 4 |

+++++
 ===== NO CHANGING OF 0'S AND 1'S INTO DON'T CARES =====

MAPPINGS ASSIGNMENTS:

- 1: 0 0
- 2: 0 1
- 3: 1 1
- 4: 1 0

----- PRE-NODE -----

P= A

Q= B + A

----- TREE-NODE -----

P= Q1*P0' + P1*P0

Q= Q1*Q0' + P1*Q0

----- OUT-NODE -----

X= Q*X(-1)' + P*X(-1)

COMPARATOR (>,<). X(I) = X(I-1) + A(I)*B(I)*Y(I-1)', Y(I) = Y(I-1) + A(I)*B(I)*X(I-1)'

MAPPINGS TABLE:

| | 1 | 2 | 3 | ! |
|---|---|---|---|---|
| 1 | ! | 1 | 2 | 3 |
| 2 | ! | 2 | 2 | 2 |
| 3 | ! | 3 | 3 | 3 |

STATES' CODES:

1: 0 0
2: 0 1
3: 1 0

MAPPINGS' NATURAL ASSIGNMENTS:

1: 0 0 OR 1 1
2: 0 1
3: 1 0

MULTIPLICATION TABLE:
(3 BUNCHES)

| | 1 | 2 | 3 |
|---|---|---|---|
| 3 | ! | 3 | 3 |
| 2 | ! | 2 | 2 |
| 1 | ! | 1 | 2 |

$$\underline{B_4(1)}$$

- 61 -

===== 0'S --> DON'T CARES, THEN 1'S --> DON'T CARES =====

MAPPINGS ASSIGNMENTS:

1: 0 0
2: - 1
3: 1 0

----- PRE-NODE -----

P= A*B'

Q= A'*B

----- TREE-NODE -----

P= P0 + P1

Q= Q1*P0' + Q0

----- OUT-NODE -----

X= P*Q'*Y(-1)' + X(-1)

Y= Q*X(-1)' + Y(-1)

$$\underline{B_4(1)}$$

SECOND ORDER EXAMPLE. $X(I) = A(I)*X(I-1)'*X(I-2) + B(I)*X(I-1)$

MAPPINGS TABLE:

| | ! | 1 | 2 | 3 | 4 | ! |
|---|---|---|---|---|---|---|
| 1 | 1 | ! | 1 | 1 | 1 | ! |
| 1 | 2 | ! | 1 | 1 | 2 | ! |
| 2 | 1 | ! | 1 | 2 | 1 | ! |
| 2 | 2 | ! | 1 | 2 | 1 | ! |

STATES' CODES:

- 1: 0
- 2: 1

MAPPINGS' NATURAL ASSIGNMENTS:

- 1: 0 0
- 2: 0 1
- 3: 1 0
- 4: 1 1

$$\underline{B_3(2)}$$

MULTIPLICATION TABLE:
(12 BUNCHES)

| | | | | | |
|-------|---|---|---|---|---|
| 4, 1 | ! | 1 | 4 | 1 | 4 |
| 3, 1 | ! | 1 | 3 | 1 | 3 |
| 4, 4 | ! | 1 | 4 | 1 | 4 |
| 1, 4 | ! | 1 | 1 | 4 | 4 |
| 4, 3 | ! | 1 | 4 | 1 | 4 |
| 2, 3 | ! | 1 | 2 | 3 | 4 |
| 3, 3 | ! | 1 | 3 | 1 | 3 |
| 1, 3 | ! | 1 | 1 | 3 | 3 |
| 2, 2 | ! | 1 | 2 | 1 | 2 |
| 1, 2 | ! | 1 | 1 | 2 | 2 |
| 2, 1 | ! | 1 | 2 | 1 | 2 |
| 1, 1 | ! | 1 | 1 | 1 | 1 |
| ----- | | | | | |
| | ! | 1 | 2 | 3 | 4 |

ZERO-LEVEL CONVERSION TABLE:

| | | |
|------|-----|------|
| 1, 1 | --> | 1, 1 |
| 2, 1 | --> | 1, 1 |
| 3, 1 | --> | 2, 1 |
| 4, 1 | --> | 2, 1 |
| 1, 2 | --> | 1, 2 |
| 2, 2 | --> | 2, 2 |
| 3, 2 | --> | 1, 2 |
| 4, 2 | --> | 2, 2 |
| 1, 3 | --> | 1, 3 |
| 2, 3 | --> | 3, 3 |
| 3, 3 | --> | 2, 3 |
| 4, 3 | --> | 4, 3 |
| 1, 4 | --> | 1, 4 |
| 2, 4 | --> | 4, 4 |
| 3, 4 | --> | 1, 4 |
| 4, 4 | --> | 4, 4 |

+++++ USING NATURAL ASSIGNMENT +++++

===== NO CHANGING OF 0'S AND 1'S INTO DON'T CARES =====

MAPPINGS ASSIGNMENTS:

1: 0 0
2: 0 1
3: 1 0
4: 1 1

----- PRE-NODE -----

E= A

F= B

----- ZERO-LEVEL-NODE -----

PART 1 OF THE ZERO-LEVEL-NODE:

P= F2*E1

Q= E2*F1' + F2*F1

----- TREE-NODE -----

P= F2*F1'*F0 + Q2*F1

Q= F2*Q1'*Q0 + Q2*Q1

----- OUT-NODE -----

X= P*X(-1)'*X(-2) + Q*X(-1)

List of References

- [1] Gajski, D. D., "Semigroups of Recurrences," High Speed Computer and Algorithm Organization, Academic Press, 1977, pp. 179-183.

- [2] Gajski, D. D., "Recurrence Semigroups and Their Relation to Data Storage in Fast Recurrence Solvers on Parallel Machines," University of Illinois at Urbana-Champaign, Department of Computer Science, Report UIUCDCS-R-80-1037, September 1980.

- [3] Kohavi, Z., Switching and Finite Automata Theory, McGraw-Hill, 1970, pp. 90-102.

| | | | |
|---|------------------------------------|--|---|
| BIBLIOGRAPHIC DATA SHEET | 1. Report No. UIUCDCS-R-80-1040 | 2. | 3. Recipient's Accession No. |
| 4. Title and Subtitle AUTOMATIC CELL GENERATION FOR RECURRENCE STRUCTURES | | | 5. Report Date November 1980 |
| | | | 6. |
| 7. Author(s) Avinoam Bilgory and Daniel D. Gajski | | | 8. Performing Organization Rept. No. UIUCDCS-R-80-1040 |
| 9. Performing Organization Name and Address University of Illinois at Urbana-Champaign Department of Computer Science Urbana, Illinois 61801 | | | 10. Project/Task/Work Unit No. |
| | | | 11. Contract/Grant No. US NSF MCS76-81686 |
| 12. Sponsoring Organization Name and Address National Science Foundation Washington, D. C. 20550 | | | 13. Type of Report & Period Covered Technical Report |
| | | | 14. |
| 15. Supplementary Notes | | | |
| 16. Abstracts The cell generation module of a Gate-to-Silicon compiler is described. In this paper the cell generation is restricted to Boolean recurrence structures which appear to be the most difficult for synthesis. The fastest solution of the recurrence can be achieved by the semigroups' approach in time proportional to the logarithm of the recurrence length. The solution is accomplished by a network that requires up to 4 different types of cells. Given a Boolean recurrence of any order, the cell generator module generates the Boolean equations of these cells. | | | |
| 17. Key Words and Document Analysis. 17a. Descriptors Gate compilers Logic-design automation Boolean-recurrence solvers | | | |
| 17b. Identifiers/Open-Ended Terms | | | |
| 17c. COSATI Field/Group | | | |
| 18. Availability Statement RELEASE UNLIMITED | | 19. Security Class (This Report) UNCLASSIFIED | 21. No. of Pages 73 |
| | | 20. Security Class (This Page) UNCLASSIFIED | 22. Price |

UNIVERSITY OF ILLINOIS-URBANA



3 0112 028215066